RBE595 Deep Learning for Advanced Robot Perception: HW3

Abdelhamid
Dep. of Robotics Engineering
Worcester Polytechnic Institute

***Abstract*— A sectioned case study of two discriminative learning algorithms (MLP and CNN) applied to a supervised learning classification problem using the MNIST digits dataset. For each of the two algorithms, this report outlines the development efforts (and resulting performance improvements) implemented on the path to less than 1% prediction error.**

MNIST is a collection of 70K total images predefined into 60K training images and 10K testing images. The user is free to use some or all of the training images to build their model and they can use some or all of the testing data to test their model. Here we use the first 10K training images and the first 1K testing images. Each image consists of a 28x28 matrix with each cell containing the value of the corresponding pixel (0-255). Of course, the 28x28 matrices need to be "flattened" or reshaped into a 1x784 vector so that each image can be fed to the network by way of each image pixel corresponding to one of the input neurons/perceptrons.

This report aims to first plot and observe the behavior of the baseline models over enough epochs and then use that to guide the optimization efforts. The expectation is that the reasoning behind each addition/change to the baseline model is stated and then that it's effect on the performance of the model (either positively or negatively) is analyzed and justified. Improvements and changes range from simple changes as varying the number of epochs and/or batch size to more complex developments such as changing the network size/structure and and/or adding dropout layers.

Finally, as a matter of reference, the term "error" is used in this report to represent "prediction error" of the validation/test dataset, exclusively. In other words, this is NOT to be confused with the prediction error observed on the train dataset. Furthermore, this "error" term is completely separate from the "loss" values discussed in the report.

I MULTI LAYER PERCEPTRON (MLP)

As Minsky and Papert argued in their 1969 book *Perceptrons*, a simple perceptron can only solve linearly separable problems. In other words, if we were to visualize the data, it can be separated either by a line (2D) or a plane (3D). However, most interesting problems, such as digit classification, involve datasets that are not linearly separable. For more complex problems, an MLP offers an endless range of solutions given different combinations of simple perceptrons connected over several layers (single input, variable hidden, single output) resembling a network.

Algorithm Outline

- Fix numpy random seed generator to predefined scalar in order to avoid numpy defaulting to use the time stamp as the seed
- Load data using the mnist library *load_data()* function. Note that for the remainder of this report, X refers to instances and Y refers to labels. This is an important distinction when considering X and Y for both train and test datasets
- For using the categorical cross entropy loss function (where each instance can belong to one class only) along with a softmax activation function, we need our list of labels (Y) to be in the form of a categorical matrix of all 0s and only 1s in the column corresponding to each class. More specifically, instead of having a single column of labels 0-9, we need an Nx10 matrix consisting of 10 columns corresponding to each of the classes 0-9. The first row of this matrix is the "one-hot" encoded label representation of the first image instance in X. The reason for this is that the output layer of this network consists of 10 different neurons corresponding to the number of classes. Comparing this output to a single label will make no sense, but comparing it to another 10 bit encoding of that label works perfectly
- Defining the model consists of declaring the block structure for Keras to start building the network, then defining the layers therein, and finally compiling the network by specifying the loss function and optimizer to use and the metrics to record. This baseline model involves 3-layer network, with the input layer corresponding to the number of pixels and the output layer corresponding to the number of classes.
- Once the model is built, we can use the *fit()* function from Keras to fit the model to the data. Here we can define the number of epochs, the batch size, and the level of terminal annotation during runtime
- Finally, we evaluate the model. The baseline script deploys a print function for a somewhat live-feed of the model as it flows through the epochs. Here, we add a set of plotting functions (using pyplot from the matplotlib library) in order to better visualize the model accuracy/loss on both the train and the validation datasets as shown in Figures 1 and 2 below

Baseline Performance (1.79% Error)

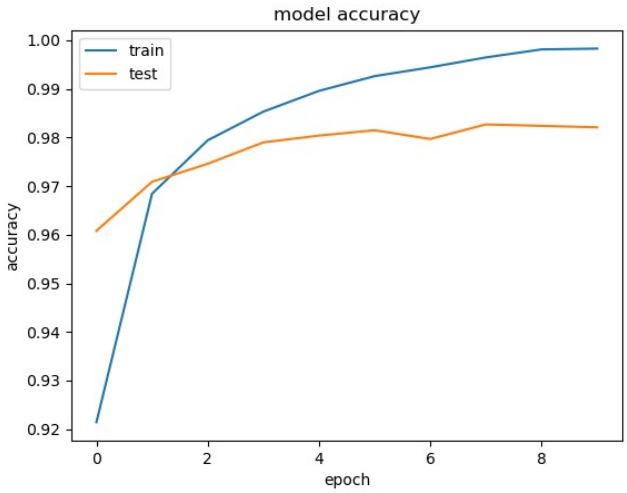

Figure 1 : MLP Baseline Accuracy

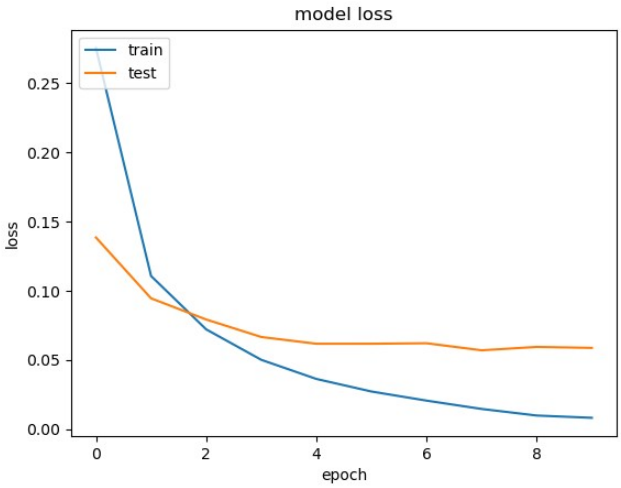

Figure 2 : MLP Baseline Loss

Optimization

Now that we have the baseline performance plotted, we can better understand the behavior of the model and can then start to formulate an opinion on the fit. In Figure 1 we can see the training accuracy is seen to rapidly surpass that of the validation accuracy after only two epochs. Looking further down the epoch lifetime we can see that the validation accuracy begins to plateau almost halfway through. Both observations point to overfitting and to the fact that the later epochs are simply wasted computer processes. However, upon closer observation we can see that both the train and validation accuracies are very high, the difference is only exacerbated by the display range of the y-axis.

Given that, we can tell that the model is slightly overfitting but not so much so that adding complexity to the network will further increase this phenomenon. In fact, referring to the defining class of the baseline model we can see that the network is currently extremely basic with only one hidden layer and no dropout utilization. As we learned through multiple implementations of deep ("large") vs shallow networks we know that there is a benefit in adding more layers when it comes to extracting features. More hidden layers offer higher levels of abstraction and is what we currently need to help our model better generalize on the validation dataset. In addition, another form of increasing complexity is increasing the number of nodes/neurons per layer so we will also be experimenting with this. For each hidden layer we also reserve the possibility of treating it with a predefined dropout percentage.

Furthermore, we will also be making use of certain Keras tools such as weight/kernel initialization in order to specify an initial set of weights for each layer (more points of experimentation); and such as batch/layer normalization in order to normalize the output coming out of each layer as to prevent one weight from growing too large and skewing the input of the next layer. Note that we can also be using the kernel constraint argument but for this dataset we observed that layer normalization works better than both batch normalization and kernel constraints.

Note the lack of changes in either the optimizer type or the activation functions of the layers given that the adam optimizer is proven to be the best (especially given the built in weight decay and momentum/beta_1) and the relu activation function better preserves the features in the images. For example, after already normalizing the pixel values, using an activation function such as the sigmoid will cause further resolution loss. Both adam and relu are already part of the baseline model and are only expanded upon (such as by a, not replaced. Another factor that was tested but avoided early on is the use of regularization as all forms (kernel, layer, or output) have proven to be less effective than layer normalization.

The following is the list of changes/additions and reasoning thereof:

1. Added 3 more hidden layers with significant number of neurons (600, 300, 200) keeping the same input and output layers. This was the first clear step in the direction of better accuracy especially given the extremely high accuracy of the dense neural network developed by Meier et. al. With a powerful enough GPU, they were able to achieve 0.35% error using a 6 layer (2500, 2000, 1500, 1000, 500, 10) network and therefore we know that there is benefit to increasing the model complexity through added layers and neurons
2. Added layer normalization after every layer in order to normalize the inputs to every layer, further reducing the noise. Note that the image pixel values are already normalized and therefore there is no need to specify normalization for the input values of the input layer
3. Added weight initiation by using an orthogonal function to define the kernel initialization argument for each layer. Experimented with he_normal and random normal initializers but otthogonal was a clear best
4. Added dropout (0.3) to the visible layer as to act as one form of data augmentation. This way, the first hidden layer will receive representations of images that are slightly altered and will help in preventing overfitting and will promote generalization

Final Results (1.02% Error)

When it comes to improving model performance by a fraction of a percent (1.7% to 1% error), even extremely complex and modified models tend to require a dramatic increase in the number of epochs in order to reach the desired accuracy. With 300 epochs, Figures 3 and 4 below show the performance of the best model we were able to achieve.

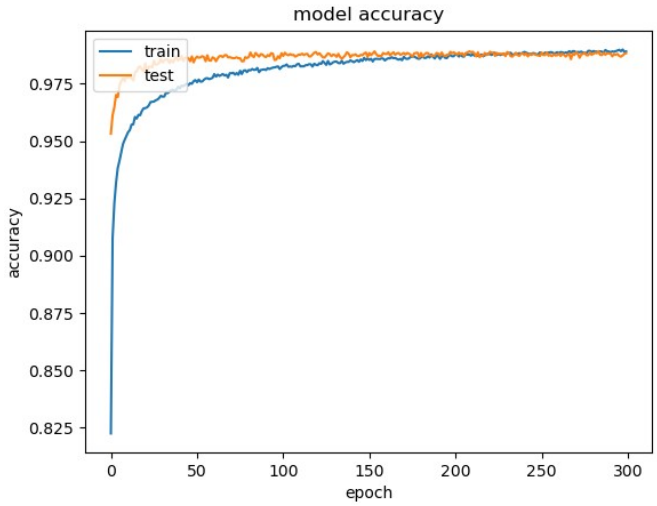

Figure 3 : MLP Improved Accuracy

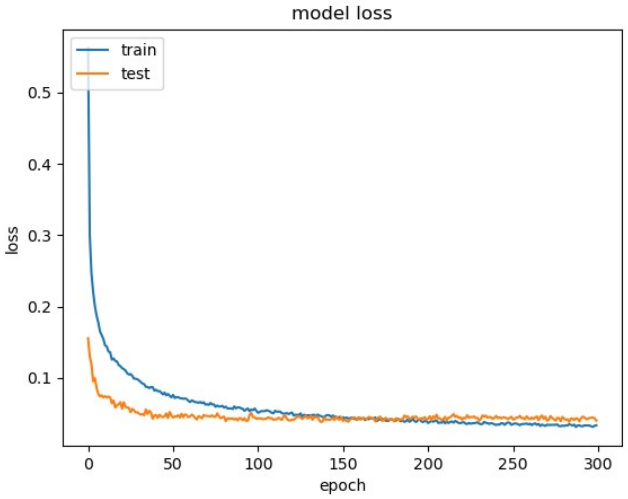

Figure 4 : MLP Improved Loss

II Convolution Neural Network (CNN)

MLPs and CNNs are both Neural Networks but they each have vastly differing architectures. Here is an outline of the main differences between the two networks:

- MLPs are densely connected which means that every single neuron from a given layer is connected to every single neuron from the following layer. This layout results in a massive number of weights to train. In contrast, CNNs are sparsely connected with each node in a given convolution layer receiving an input from a small number of nodes in the previous layer (its receptive field) resulting in a low number of weights compared to MLP
- CNNs also benefit from parameter sharing where each member of the convolution kernel (filter) is used at every pixel/node from the input/previous layer. In MLPs, each pixel/input has its own dedicated weight which again makes CNNs far more efficient in terms of computation given the even further reduction in number of parameters
- The output from each layer of an MLP is a single representation of the previous layer activation functions. In the case of CNNs, the output of a convolution layer are multiple representations known as feature maps. The number of feature maps is defined by the number of kernels used
- CNNs can operate on 2D format inputs, MLPs require pixel flattening. This not only makes CNNs easier to use, but it also makes them more accurate seeing as the spatial features are better preserved. This paired with the receptive field argument above, make CNNs far more capable than MLPs

Algorithm Outline

- Fix numpy random seed generator to predefined scalar in order to avoid numpy defaulting to use the time stamp as the seed
- Load data using the mnist library *load_data()* function. Note that for the remainder of this report, X refers to instances and Y refers to labels. This is an important distinction when considering X and Y for both train and test datasets
- Reshape images (X) in the form of samples, channels, width, and height so as to fit in the 28X28 format. Note how there is no need to "flatten" the data seing as Keras allows for the direct use of 2 Dimensional convolution functions.
- Defining the model consists of declaring the block structure for Keras to start building the network, then defining the layers therein, and finally compiling the network by specifying the loss function and optimizer to use and the metrics to record. This baseline model involves a 28x28 input layer, a single convolutional layer with 32 kernels of size 5x5 followed by a maxpooling layer of size 2x2, and finally a densely connected layer between the maxpooling layer and the output layer
- Note the need to flatten the output from the maxpooling layer prior to feeding it to the dense layer given that we know the dense layer will be looking for input from a flat list of neurons
- Once the model is built, we can use the *fit()* function from Keras to fit the model to the data. Here we can define the number of epochs, the batch size, and the level of terminal annotation during runtime
- Finally, we evaluate the model. The baseline script deploys a print function for a somewhat live-feed of the model as it flows through the epochs. Here, we add a set of plotting functions (using pyplot from the matplotlib library) in order to better visualize the model accuracy/loss on both the train and the validation datasets as shown in Figures 1 and 2 below

Baseline Performance (1.11% Error)

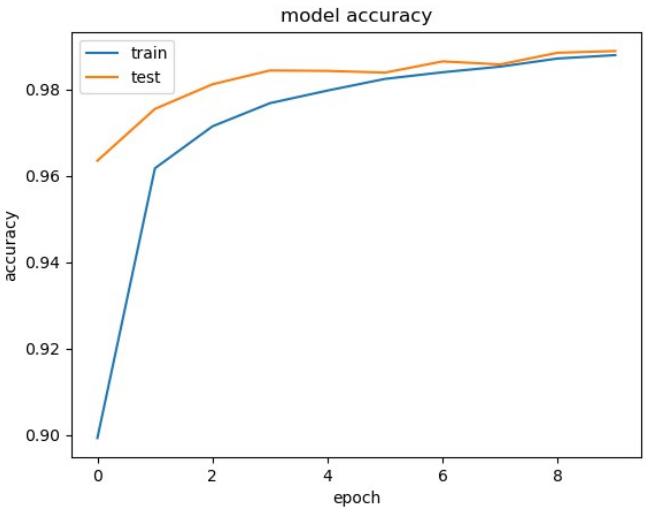

Figure 5 : CNN Baseline Accuracy

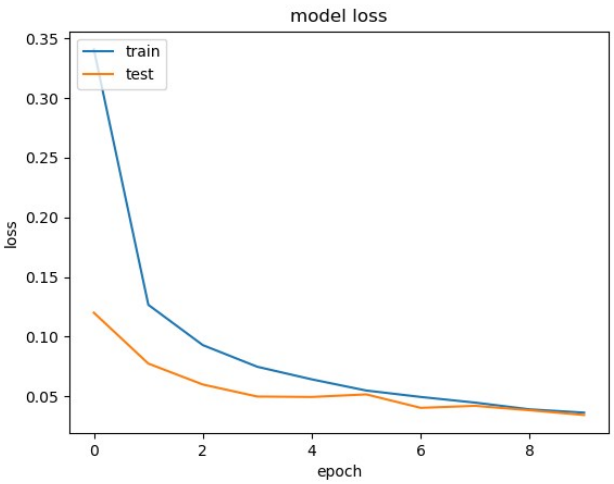

Figure 6 : CNN Baseline Loss

Optimization

Of course, we can already see that the baseline model for the CNN is clearly surpassing that of the baseline MLP and is actually fairly close to the "brute force" improvemed MLP model. Here we use some of the knowledge we gained from the previous section to guide our efforts.

Again, we held the reul activation functions, the adam optimizer, and the learning rate constant, with the following changes implemented:

1. Before going on to add more layers to the system in pursuit of a higher complexity, a better approach would be to first increase the number of neurons for the available layer and evaluating. Increasing the number of neurons to 400 without any more dense layers proved sufficient
2. Added weight initiation by using an orthogonal function to define the kernel initialization argument for each layer. Experimented with he_normal and random normal initializers but otthogonal was a clear best
3. Increase dropout rate to 0.5 since the optimal range for hidden layer dropout seems to be around 0.3 to 0.5 with the baseline 0.2 being too low and resulting in overfitting
4. Increase number of epochs to 50 to allow the model to converge. Note how it is now taking longer for the train accuracy to catch up to the validation accuracy which is a great sign that our model is not suffering from severe overfitting
5. Increased batch size to 200 given the higher baseline accuracy of the CNN, increasing the batch size will improve compute efficiency without severely impacting the generalization error

Final Results (0.86% Error)

As was expected, it is almost effortless to improve the performance of a CNN especially compared to MLPs. In just a fraction of the epochs and an even smaller fraction of the number of parameters, a CNN is able to achieve sub 1% error with ease as shown in figures 7 and 8 below.

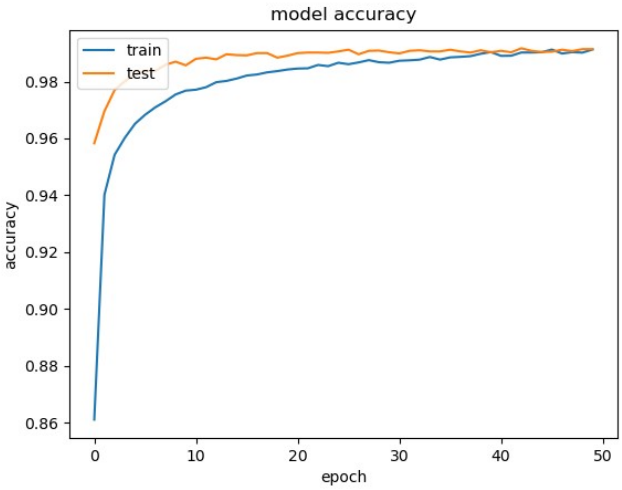

Figure 7 : CNN Improved Accuracy

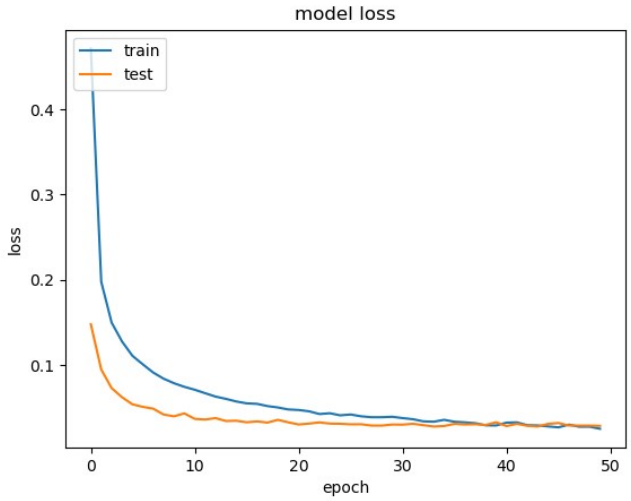

Figure 8 : CNN Improved Loss

III REFERENCES

1 C. Morato. RBE595 ST. Deep Learning for Advanced Robot Perception Lecture Series, Robotics Engineering Dept. Worcester Polytechnic Institute, Worcester, MA, Sep. – Dec. 2021.

2 Kevin P. Murphy, *Machine Learning, a Probabilistic Perspective*, 1st ed. Cambridge, MA: MIT Press 2012.

3 J. Widom, "Tips For Writing Technical Papers," *Stanford InfoLab*, January, 2006. [Online serial] Available: https://cs.stanford.edu/people/widom/paper-writing.html [Accessed: Dec. 2, 2020].

4 M. Minsky, S. Papert, and Bottou Leon, *Perceptrons: An Introduction to Computational Geometry.* Cambrige MA: MIT Press, 2017

5 D. C. Cireşan, U. Meier, L. M. Gambardella and J. Schmidhuber, "*Deep, Big, Simple Neural Nets for Handwritten Digit Recognition,*" in *Neural Computation*, vol. 22, no. 12, pp. 3207-3220, Dec. 2010, doi: 10.1162/NECO_a_00052.